DelphiDabbler Resource File Classes

User Guide

Overview

These classes support read and writing 32 bit binary resource files. Some routines are also included that assist in manipulating resource identifiers.

The code encapsulates the high-level structure of a resource file and its resources. It deals only with raw resource data - the actual format of the raw data depends on the resource type. This code does not provide any support for specific resource types or their data formats.

This document is divided into various sections:

- Classes - description of the provided classes
  - TPJResourceFile - details of usage, methods and properties of the resource file class
  - TPJResourceEntry - details of usage, methods and properties of the resource entry class
  - EPJResourceFile - description of this exception class
- Routines - description of the helper routines
- Constants - description of constants defined for use with the classes
- Examples - examples of using the classes
  - #1 - Loading a resource file
  - #2 - Accessing all resources in a file
  - #3 - Finding a resource
  - #4 - Listing specific resources from a file
  - #5 - Adding or modifying a resource's data
  - #6 - Adding a new resources to a file
  - #7 - Checking is a resource exists
  - #8 - Deleting resources
  - #9 - Saving a resource file
  - #10 - A practical example
- Appendix: Resource File Structure Notes

Classes

The classes included in this release are as follows:

Class	Description
TPJResourceFile	Encapsulates a 32 bit binary resource file and exposes the entries within it. The class allows resources within the file to be accessed, created, read and searched for.
TPJResourceEntry	Encapsulates a single resource within a resource file. Permits access to the resource type, name and language. Other header data can be modified. Provides read/write access to the raw resource data through a *TStream*.
EPJResourceFile	Class of exception raised by the above classes to report errors.

Although we use the word “file” in these notes, this term also covers binary resource data stored in a stream.

Detailed descriptions of the classes now follow.

TPJResourceFile

This class is used to encapsulate a 32 bit resource file, to find which resources in contains and to add new and delete existing resources. The number of resources in the file is given by the *EntryCount* property while the *Entries[]* property provides access to them. Resource entries are not created directly by the user but via the *AddEntry()* method of this class.

Resource entries accessed via this class are *TPJResourceEntry* objects and have methods and properties that can be used to interrogate and update them.

Methods

`constructor Create;`

Class constructor. Creates a new empty resource file object.

`destructor Destroy;`

Class destructor. Usually called via the *Free* method. Destroying the object frees all the resource entries currently in the file.

`procedure Clear;`

Clears all resources from the file object. The resources are freed.

`function DeleteEntry(const Entry: TPJResourceEntry): Boolean;`

Deletes a resource entry from the resource file object if it exists. The entry object is not freed and must be freed by the user since it will no longer be freed by the *Clear* or *Destroy* methods. The recommended way to delete and free a resource entry is to free the entry object since this automatically unlinks it from the resource.

- Parameter: *Entry* - reference to resource entry object to be deleted.
- Returns true if entry was in the resource list and was deleted and false otherwise.

```
function IndexOfEntry(const Entry: TPJResourceEntry): Integer;
```

Gets the index number of the given resource entry in the resource file's *Entries[]* property.

- Parameter: *Entry* - reference to resource entry to be found.
- Returns index number if entry was found and -1 if entry is not in the resource file.

```
procedure LoadFromFile(const FileName: TFileName);
```

Loads a resource file from the named file, replacing any existing resource.

- Parameter: *FileName* - the name of the file from which to load the resource
- Exceptions are raised if the file does not exist or does not contain a valid resource.

```
procedure LoadFromStream(const Stm: TStream);
```

Loads a resource "file" from the current location in the given stream.

- Parameter: *Stm* - the stream from which to load the resource
- Exceptions are raised if the stream does not contain a valid resource file.

```
procedure SaveToFile(const FileName: TFileName);
```

Saves data as a 32 bit resource file.

- Parameter: *FileName* - the name of the file.
- Exceptions are raised if the file cannot be created.

```
procedure SaveToStream(const Stm: TStream);
```

Saves data in resource file format on the given stream at the current location.

- Exceptions are raised if the stream does not support writing.

```
class function IsValidResourceStream(const Stm: TStream): Boolean;
```

Checks if the given stream contains data representing a valid 32 bit resource file starting at the current location. This method checks that a 32 bit resource file header is present but does not validate the whole of the file. Note that the stream is not rewound to the starting position after the check is made.

- Parameter: *Stm* - the stream containing the resource file.

```
function AddEntry(const ResType, ResName: Pchar;
  const LangID: Word = 0): TPJResourceEntry;
```

Adds a new, empty, resource to the current "file" object.

- Parameter: *ResType* - the type of the new resource (ordinal or string).
- Parameter: *ResName* - the name of the new resource (ordinal or string)
- Parameter: *LangID* - optional language id of the resource (a language neutral value of 0 is used if this parameter is not provided).
- Returns a reference to the new resource entry. This reference should be used to set the resource headers and to store the raw data.
- Exception raised if an entry already exists with same type, name and language id.

```
function AddEntry(const Entry: TPJResourceEntry; const ResName: Pchar;
  const LangID: Word = 0): TPJResourceEntry;
```

Adds a copy of the given resource entry to the current "file" object with a new resource name and language id. The new entry has the same resource type as the one being copied.

- Parameter: *Entry* - reference to the resource entry to be copied.
- Parameter: *ResName* - name of the new resource.
- Parameter: *LangID* - optional language id of the resource (a language neutral value of 0 is used if this parameter is not provided).
- Returns a reference to the new resource entry that has the same header information and data as the one being copied except for resource name and language id.
- Exception raised if an entry already exists with same type, name and language id.

```
function FindEntry(const ResType, ResName: Pchar;
  const LangID: Word = $FFFF): TPJResourceEntry;
```

Finds a resource entry with the given type, name and language id. The search can ignore the resource name and or language id in which case first entry that matches the provided information is found.

- Parameter: *ResType* - the type of the resource to be found (ordinal or string) - required.
- Parameter: *ResName* - the name of the resource to be found (ordinal or string). If just the first resource of the given type is required then *nil* can be specified here.
- Parameter: *LangID* - the language id of the required resource (optional). If only the first matching resource for the given type and name is required this parameter can be left out (or $FFFF supplied). To find a language neutral resource specify 0 for this parameter.
- Returns a reference to the found resource or *nil* if no resource was found.

```
function FindEntryIndex(const ResType, ResName: Pchar;
  const LangID: Word = $FFFF): Integer;
```

Finds the *Entries[]* property index of a resource entry with the given type, name and language id. The search can ignore resource name or language id in which case first entry that matches the provided information is found.

- Parameter: *ResType* - the type of the resource to be found (ordinal or string) - required.
- Parameter: *ResName* - the name of the resource to be found (ordinal or string). If just the first resource of the given type is required then *nil* can be specified here.
- Parameter: *LangID* - the language id of the required resource (optional). If only the first matching resource for the given type and name is required this parameter can be left out (or $FFFF supplied). To find a language neutral resource specify 0 for this parameter.
- Returns the index of the found resource in the *Entries[]* property or -1 if no resource was found.

```
function EntryExists(const ResType, ResName: Pchar;
  const LangID: Word = $FFFF): Boolean;
```

Checks whether a resource entry matching given search criteria exists.

- Parameter: *ResType* - the type of the resource to be found (ordinal or string) - required.
- Parameter: *ResName* - the name of the resource to be found (ordinal or string). If just the first resource of the given type is required then *nil* can be specified here.

- Parameter: *LangID* - the language id of the required resource (optional). If only the first matching resource for the given type and name is required this parameter can be left out (or $FFFF supplied). To find a language neutral resource specify 0 for this parameter.
- Returns true if a resource matching the search criteria exists and false if not.

Properties

```
property EntryCount: Integer;
```

Read-only property that provides the number of resources in the resource file (i.e. the number of entries in the *Entries[]* property).

```
property Entries[Idx: Integer]: TPJResourceEntry;
```

Read-only property that provides access to all the resources in the resource file by index.

TPJResourceEntry

Encapsulates a resource within a resource file. Object of this class must not be directly instantiated. *TPJResourceFile* automatically creates the required objects when they are read from a file or can create new instances in its *AddEntry* methods. *TPJResourceFile* is actually an abstract class that provides the required interface for manipulating resource entries. The actual concrete class that implements the resource entry is private. Therefore all valid resource objects are accessed via *TPJResourceFile*.

TPJResourceEntry objects are used to manipulate and interrogate resource entries. This is done mainly via properties which give access to resource header information, allow some header information to be set and give read/write access to the raw resource data via a *TStream*.

Care should be taken not to use a resource entry after the resource file object has been cleared or destroyed or a new resource file has been read since all these actions free previous resource entries. A *TPJResourceEntry* object can be freed directly - doing so removes the entry from any resource file object it belongs to.

Methods

```
function IsMatching(const ResType, ResName: Pchar;
  const LangID: Word = $FFFF): Boolean;
```

Checks if the resource entry has the given type, name and language id. The resource name and/or language id can be ignored in which case only the values provided will be matched. For example to match only a resource type use `IsMatching(MyResType, nil);`

- Parameter: *ResType* - the resource type (ordinal or string) - required.
- Parameter: *ResName* - the resource name (ordinal or string) - if *nil* is passed as this parameter then it is ignored when matching.
- Parameter: *LangID* - the language id of the resource - if this is left out or $FFFF is passed then it is ignored in the match. To test for a language neutral resource specify 0 in this parameter.

```
function IsMatching(const Entry: TPJResourceEntry): Boolean;
```

Checks if the resource entry has the same type, name and language ID as the given resource entry.

- Parameter: *Entry* - the resource entry to match against.

- Returns true if the entries match and false otherwise.

Properties

`property DataSize: DWORD;`

Read-only. Provides the size of the resource data (same as calling *Data.Size*).

`property HeaderSize: DWORD;`

Read-only. Provides the size of the resource header (which varies depending on the type and size of the resource type and name).

`property DataVersion: DWORD;`

Gets or sets the predefined data resource version information.

`property MemoryFlags: Word;`

Gets or sets the attribute bitmask that specifies the state of resource.

`property LanguageID: Word;`

Read-only. Gets the language used by the resource (value of 0 is language-neutral).

`property Version: DWORD;`

Gets or sets the user specified version number for resource data.

`property Characteristics: DWORD;`

Gets or sets the user specified characteristics of the resource.

`property ResName: PChar;`

Read-only. Gets the name of the resource. This value is either a pointer to a zero-terminated string or an ordinal value as returned from *MakeIntResource()*.

`property ResType: PChar;`

Read-only. Gets the type of the resource. This value is either a pointer to a zero-terminated string or an ordinal value as returned from *MakeIntResource()*.

`property Data: TStream;`

Read-only. Gets a reference to the stream that contains the resource's raw data. The stream can be used to read or write the data. Any padding bytes that follow the resource's data are not included in the stream.

EPJResourceFile

This is the class of exceptions that are raised directly by *TPJResourceFile* and *TPJResourceEntry*. Note that some methods may raise exceptions of other classes.

The class defines no new methods or properties.

Routines

The helper routines provided with this code can be used to assist in manipulating resource identifiers. They can be useful for working with Windows API routines as well as the classes presented here. Here are the routines.

function `IsIntResource(`**const** `ResID: PChar): Boolean;`

A clone of the `IS_INTRESOURCE` macro defined on MSDN that checks if a resource identifier is ordinal or a string. Complements the *MakeIntResource* “macro” defined in `Windows.pas`.

- Parameter: *ResID* - the resource identifier to check.
- Returns true if the identifier is numeric (as produced by *MakeIntResource*) or false if it is a pointer to a null terminated string.

function `IsEqualResID(`**const** `R1, R2: PChar): Boolean;`

Checks for equality of two resource identifiers. To be equal the identifiers either be ordinal and have the same value or must both point to strings that have the same text when case is ignored.

- Parameter: *R1* - the first resource identifier to check.
- Parameter: *R2* - the second resource identifier to check.
- Returns true if the identifiers are equal and false otherwise.

function `ResIDToStr(`**const** `ResID: PChar):` **string**`;`

Converts a resource identifier into its string representation as defined on MSDN.

- Parameter: *ResID* - the resource identifier to convert.
- Returns the string representation. If the identifier is a string pointer then the string itself is returned. If the identifier is ordinal then the returned string is the integer value preceded by a # character.

Constants

Some constants are defined to assist in setting some of the class properties.

Memory Flags Constants

The following flags are used to form the bitmask in a resource entry's *MemoryFlags* property. The first four constants in the table can or ORd together to form the bitmask. The final three constants are complements of the first three and are ANDed against the bitmask to remove their complement from the bitmask.

Constant	Value	Description
RES_MF_MOVEABLE	$0010	The system can move the resource in memory. If this flag is not present the resource is fixed in memory.
RES_MF_PURE	$0020	The resource contains DWORD aligned data so that padding is not required. If this flag is not present the resource is not DWORD aligned and must be padded.
RES_MF_PRELOAD	$0040	The resource is to be loaded in memory just after the application itself has been loaded. If not present then loading of the resource may be deferred until required by the application.
RES_MF_DISCARDABLE	$1000	If set then on low memory conditions, the resource can be removed from memory and then reloaded when the application needs it, otherwise the resource must remain in memory.
RES_MF_FIXED	$FFEF	Complement of RES_MF_MOVEABLE: used to remove this flag from the bitmask.
RES_MF_IMPURE	$FFDF	Complement of RES_MF_IMPURE: used to remove this flag from the bitmask.
RES_MF_LOADONCALL	$FFBF	Complement of RES_MF_LOADONCALL: used to remove this flag from the bitmask.

Note that Windows NT ignores RES_MF_MOVEABLE, RES_MF_IMPURE and RES_MF_PRELOAD.

Predefined Resource Types

Delphi's Windows unit defines all the predefined resources types known at the time of writing except RT_HTML and RT_MANIFEST. Therefore these two type identifiers are defined in this unit for convenience:

```
RT_HTML = MakeIntResource(23);
RT_MANIFEST = MakeIntResource(24);
```

See Appendix 1 for a description of all the predefined resource types.

Examples

Example 1: Loading a resource file

In this first example we demonstrate how to create a resource file object and how to load a file into it. The following code fragment shows how this is done.

```
var
  ResFile: TPJResourceFile;
  ...
begin
  ResFile := TPJResourceFile.Create;
  try
    ResFile.LoadFromFile('MyResFile.res');
    ...
    // Do something with resource file object
    ...
  finally
    ResFile.Free;
  end;
end;
```

First we create a *TPJResourceFile* object and then use its *LoadFromFile* method read a file from disk. We then process the file in some way and once we are finished we free the resource file object. That's all there is to it. Note that if the given file does not contain a valid 32 bit resource file an exception will be raised.

We can also read resource data from a stream rather than loading from file by using the *LoadFromStream* method of *TPJResourceFile* in place of *LoadFromFile*.

Example 2: Accessing all resources in a file

In our next example we show how to scan through all the resources in a file and how to list some information about each one. The following example assumes we have created a resource files object *ResFile* and have loaded a resource file into it (as in example 1). We also assume that the form contains a memo named *Memo1*. Here is the code:

```
var
  ResFile: TPJResourceFile;
  ResEntry: TPJResourceEntry;
  EntryIdx: Integer;
begin
  ...
  // Assume ResFile contains a loaded resource file
  ...
  Memo1.Clear;
  for EntryIdx := 0 to Pred(ResFile.EntryCount) do
  begin
    ResEntry := ResFile.Entries[EntryIdx];
    Memo1.Lines.Add(
      Format(
        'Type: "%s"   Name: "%s"   LanguageID: %0.4X',
        [ResIDToStr(ResEntry.ResType), ResIDToStr(ResEntry.ResName),
        ResEntry.LanguageID]
      )
    );
  end;
  ...
  // Don't forget to free ResFile at some stage.
end;
```

This code uses both *TPJResourceFile* and *TPJResourceEntry* objects. The resource file's *EntryCount* property tells us how many resources there are in the file. Each resource is represented by a *TPJResourceEntry* object made available from the integer indexed *Entries[]* array property. So we loop through all the valid indexes in *Entries[]* and store a reference to each resource entry in turn. Having got the resource entry object we now access its *ResType*, *ResName* and *LanguageID* properties to get the information we want to display.

The details of each entry are formatted by Delphi's *Format* function and added to *Memo1*. Note that we use the *ResIDToStr* helper function to get a string representation of the resource type and name. We display the language ID as a four digit hex number since the value is a `Word`.

You may have noticed that we have not freed any of the resource entry objects. This is not necessary since they are all freed automatically when the resource files object is freed (as shown in example 1).

Example 3: Finding a resource

To find a resource we use either the *FindEntry* or *FindEntryIndex* methods. The difference is that *FindEntry* returns the *TPJResourceEntry* object for the entry (or *nil* if not found) while *FindEntryIndex* returns the index of the entry in the *Entries[]* property.

Let's assume a resource file is loaded into the *TPJResourceFile* variable *ResFile*. We want to find a `RT_HTML` resource named `INDEX_HTML`. The following code checks if such a resource exists and displays its data size in a message box, or a message saying the resource doesn't exist. This first version of the code uses *FindEntry*:

```
// Version using FindEntry
var
  ResFile: TPJResourceFile;
  Entry: TPJResourceEntry;
begin
  ...
  // Assume ResFile contains a loaded resource file
  ...
  Entry := ResFile.FindEntry(RT_HTML, 'INDEX_HTML', $0809);
  if Assigned(Entry) then
    ShowMessageFmt(
      'Data size for INDEX_HTML is %d',
      [Entry.DataSize]
    )
  else
    ShowMessage('Can''t find resource');
  ...
  // Don't forget to free ResFile at some stage.
end;
```

This second version of the code shows how the same result is obtained with *FindEntryIndex*:

```
// Version using FindEntryIdex
var
  ResFile: TPJResourceFile;
  Entry: TPJResourceEntry;
  Idx: Integer;
begin
  ...
  Idx := fResFile.FindEntryIndex(RT_HTML, 'INDEX_HTML');
  if Idx >= 0 then
  begin
    Entry := fResFile.Entries[Idx];
    ShowMessageFmt(
      'Data size for INDEX_HTML is %d', [Entry.DataSize]
    );
  end
  ...
end;
```

Note that we have used the “short form” of the *FindEntry* and *FindEntryIndex* methods above: they find the first resource with the given type and name, irrespective of language. The long version of the methods finds a specific resource type, name and language. For example the following code finds a `RT_HTML` resource named `INDEX_HTML` with language $0809:

```
// "Full" version of FindEntry
var
  ResFile: TPJResourceFile;
  Entry: TPJResourceEntry;
begin
  ...
  Entry := ResFile.FindEntryIndex(RT_HTML, 'HTMLRES_HTML', $0809);
  if Assigned(Entry) then
    ... etc ...
```

```
end;
```

Example 4: Listing specific resources from a file

We can use the *IsMatching* method of *TPJResourceEntry* to check if a specific resource matches given criteria. *IsMatching* can match just a resource type, and resource type and name or can uniquely identify a resource in a file by matching its type, name and language. Like *TPJResourceFile.FindEntry*, the language ID parameter is optional. The resource name parameter can be *nil* if we don't want to specify the name in the match.

Given the above description of *IsMatching*, we can list all the `RT_HTML` resources in a resource file in a *TMemo* with this code:

```
var
  ResFile: TPJResourceFile;
  Entry: TPJResourceEntry;
  Idx: Integer;
begin
  ...
  // Assume ResFile contains a loaded resource file
  ...
  for Idx := 0 to Pred(ResFile.EntryCount) do
  begin
    Entry := ResFile.Entries[Idx];
    if Entry.IsMatching(RT_HTML, nil) then
      Memo1.Lines.Add(
        Format('%s', [ResIDToStr(Entry.ResName)])
      );
  end;
  ...
end;
```

To list only all the different language versions of the `RT_HTML` resource named `INDEX_HTML` we simply change the *IsMatching* method call in the for loop to:

```
      Entry.IsMatching(RT_HTML, 'INDEX_HTML')
```

Example 5: Adding or modifying a resource's data

While the code in this unit does not understand the various resource data formats it does assist in reading, adding, updating and deleting the raw resource data. The *TPJResourceEntry* class's *Data* object exposes the resource data as a *TStream* which means that we can use normal stream handling techniques to access the data.

Reading the data

The following code fragment shows how to read all the data from the resource to a buffer.

```
var
  Entry: TPJResourceEntry;
  Buf: PByte;
begin
  ...
  // Make sure Entry references a resource object
  ...
  // Create buffer of required size
  GetMem(Buf, Entry.DataSize);
  try
    // Make sure resource data stream at start
    Entry.Data.Position := 0;
    // Read all resource data into buffer
    Entry.Data.ReadBuffer(Buf^, Entry.DataSize);
    // Rewind data stream again
    Entry.Data.Position := 0;
    ...
    // Do something with Buf
    ...
  finally
    // Release buffer
    FreeMem(Buf);
  end;
end;
```

We first set the buffer to the required size using *TPJResourceEntry*'s *DataSize* property (The *Data* property's own *Size* property also gives this information). We now ensure the data stream is positioned at the start (you can't assume this!) and then read al the data into the buffer using *TStream*'s *ReadBuffer* method, and finally reposition the stream again ready for the next use. Having processed the data in the buffer in some way we finally free the buffer.

It may be more convenient to copy the data stream to another stream. The next example illustrates this by storing the resource data in a file named `ResEntry.dat`:

```
var
  Entry: TPJResourceEntry;
  FS: TFileStream;
begin
  ...
  // Make sure Entry references a resource object
  ...
  // Open stream onto new file
  FS := TFileStream.Create('ResEntry.dat', fmCreate);
  try
    // Copy resource data to file
    FS.CopyFrom(Entry.Data, 0);
    Entry.Data.Position := 0;
  finally
    // Close the file
    FS.Free;
  end;
end;
```

Here we first open a stream onto a new file. We then use *TStream*'s *CopyFrom* method to copy the whole of the resource data to the file stream. Note by specifying a size of 0 to the *CopyFrom* method, *TStream* automatically positions the resource data stream to the start and copies the whole stream, so we don't need to position it first. Once again we reset the resource data stream once we are done.

Deleting data

It is very easy to delete all the data in a resource: simply set the data stream's size to 0 as follows.

```
var
  Entry: TPJResourceEntry;
begin
  ...
  // Make sure Entry references a resource object
  ...
  Entry.Data.Size := 0;
  ...
end;
```

Note that you must set the *Size* property of the entry's *Data* property here: you can't set the *DataSize* property since it is read only.

Writing data

We can add data to an existing resource simply. Let's first look at how to overwrite the existing data and then show how to append data to an existing resource. For the purposes of this example, assume we have a user defined resource that stores some plain text. Again, Entry is a *TPJResourceEntry* object that references our resource. We will replace any existing data with the text “Hello World”.

```
var
  Entry: TPJResourceEntry;
  Text: string;
begin
  ...
  // Make sure Entry references a resource object
  ...
  // Delete any existing data
  Entry.Data.Size := 0;
  // Write the required text
  Text := 'Hello World';
  Entry.Data.WriteBuffer(Pointer(Text)^, Length(Text) * SizeOf(Char));
  // Position data ready for reading
  Entry.Data.Position := 0;
  ...
end;
```

The only point of note here is the use of one of the normal idioms is writing a string to a stream: we can't just do `WriteBuffer(Text, Length(Text))`, but must specify the start of the string (`Pointer(Text)^`) and pass the correct length: since Delphi 2009 SizeOf(Char) is 2!

Now let's look at how we add more text to the end of the resource data:

```
var
  Entry: TPJResourceEntry;
  Text: string;
begin
  ...
  // Make sure Entry references a resource object
  ...
  // Move to end of existing data
  Entry.Data.Seek(0, soFromEnd);
  // Write new text
  Text := ' From DelphiDabbler';
  Entry.Data.WriteBuffer(Pointer(Text)^, Length(Text) * SizeOf(Char));
  Text := #13#10'www.delphidabbler.com';
  Entry.Data.WriteBuffer(Pointer(Text)^, Length(Text) * SizeOf(Char));
  // Reposition stream to start
  Entry.Data.Position := 0;
  ...
end;
```

Here we move the stream pointer to the end of the stream so the text we write is appended to any existing data. Given the two examples above the data ends up holding the following two lines of text:

```
Hello World from DelphiDabbler
www.delphidabbler.com
```

Example 6: Adding a new resources to a file

We can add a new resource to an existing file using *TPJResourceFile*'s *AddEntry* method. The new resource must be uniquely named within the resource file (i.e. its combined resource type, name and language id must be unique) otherwise an exception will be raised. There are two versions of the *AddEntry* method - the first simply adds a new empty resource with zeroed header properties while the second version adds a renamed copy of an existing resource. Both versions of *AddEntry* take an optional language identifier. If this is not specified the resource is language neutral (*LanguageID* = 0).

The following code snippet adds four new resource to an existing resource file object:

1. An empty, language neutral, RCDATA resource with ordinal name 42. The *MemoryFlags* property is then set to “Discardable” and its data is set to “Hello World”
2. A new language neutral RCDATA resource that is a copy of the first entry but named FORTYTWO. This resource has the same *MemoryFlags* and *Data*.
3. A new RCDATA resource also named FORTYTWO but with language ID of $0809. Again this resource is a copy of the first resource.
4. A new empty RCDATA resource named 42 with language ID of $0809.

```
var
  ResFile: TPJResourceFile;
  Entry1, Entry2,
  Entry3, Entry4: TPJResourceEntry;
const
  s42 = 'FORTYTWO';
  ord42 = MakeIntResource(42);
  sHello = 'Hello World';
begin
  ...
  // Assume ResFile references a valid object
  ...
  // Create 1st entry: empty
  Entry1 := fResFile.AddEntry(RT_RCDATA, ord42);
  // now set mem flags and data
  Entry1.MemoryFlags := RES_MF_DISCARDABLE;
  Entry1.Data.WriteBuffer(Pointer(sHello)^, Length(sHello) * SizeOf(Char));
  Entry1.Data.Position := 0;
  // Create 2nd entry as copy of entry 1
  Entry2 := fResFile.AddEntry(Entry1, s42);
  // Create 3rd entry as copy of entry 1 with language id
  Entry3 := fResFile.AddEntry(Entry1, s42, $0809);
  // Create 4th entry: empty with language id
  Entry4 := fResFile.AddEntry(RT_RCDATA, ord42, $0809);
  ...
  // Do something with the entries
  ...
end;
```

The entries we have created have the following properties:

	Entry 1	**Entry 2**	**Entry 3**	**Entry 4**
Type	RT_RCDATA	RT_RCDATA	RT_RCDATA	RT_RCDATA
Name	42	FORTYTWO	FORTYTWO	42
LanguageID	0	0	$0809	$0809
MemoryFlags	$0100	$0100	$0100	0
Data	"Hello World"	"Hello World"	"Hello World"	<empty>

Example 7: Checking is a resource exists

We noted above that an exception is raised in a duplicate resource is added to a file. To prevent this we may need to check if a resource exists and we can do this with the *EntryExists* method of *TPJResourceFile* as follows:

```
var
  ResFile: TPJResourceFile;
  Entry: TPJResourceEntry;
begin
  ...
  // Assume ResFile references a valid object
  ...
  if not ResFile.EntryExists(RT_RCDATA, 'FORTYTWO', $0809) then
    Entry := ResFile.AddEntry(RT_RCDATA, 'FORTYTWO', $0809);
  ...
end;
```

Note that the language id parameter to *EntryExists* is optional and if omitted the function checks if any resource with the given type and name exists. Furthermore, specifying *nil* for the resource name makes the routine check if any resource of the given type exists. So to check if a resource file contains any RCDATA resources use:

```
if ResFile.EntryExists(RT_RCDATA, nil) then
  // We have RCDATA resources in the file
```

Example 8: Deleting resources

We can delete all resources from a resource file simply by calling the *Clear* method of *TPJResourceFile*. In addition to deleting the resources it also frees all the *TPJResourceEntry* instances.

```
var
  ResFile: TPJResourceFile;
begin
  ...
  // Assume ResFile is a valid resource file object
  ...
  // Delete all resources
  ResFile.Clear;
  ...
end;
```

A single resource can be deleted from the resource file using the *TPJResourceFile.DeleteEntry* method. This checks if the file contains the resource and deletes it if so. However, the resource entry object is not freed. While this behaviour may be useful, it is not recommended. The preferred method is simply to free the resource entry instance. Freeing a *TPJResourceEntry* object automatically removes it from the resource file.

So, to remove a single resource *ResEntry* from the resource file use the following code:

```
var
  Entry: TPJResourceEntry;
begin
  ...
  // Assume ResEntry references a valid object
  ...
  // Delete the object from its resource file
  Entry.Free;
  ...
end;
```

Example 9: Saving a resource file

Now that we have learned how to modify a resource file, it's time to know how to save it. We simply use the *SaveToFile* or *SaveToStream* methods of *TPJResourceFile*. The following code shows how to use *SaveToFile*:

```
var
  ResFile: TPJResourceFile;
begin
  // Assume ResFile references a valid object
  ...
  // Save the file to 'MyResource.res'
  ResFile.SaveToFile('MyResource.res');
end;
```

Example 10: A practical example

Having given some contrived examples of most of the functionality in the `PJResFile` unit, let us close by presenting a useful example that uses several of the methods we have reviewed.

We will create a routine that takes a list of HTML and related files and create a resource file which has a unique resource HTML resource for each file. Such resources can be used for display in Internet Explorer, using the res:// protocol. See my article "How to create and use HTML resource files" for more information on this subject.

Here is the code of the routine:

```
procedure BuildHTMLResFile(const Files: TStrings;
  const ResFileName: string);
var
  ResFile: TPJResourceFile; // res file object
  Entry: TPJResourceEntry;  // a resource entry
  ResName: string;          // a resource name
  SrcFileName: string;      // a source file name
  SrcStm: TFileStream;      // source file stream
  FileIdx: Integer;         // loops thru Files
begin
  // Create new empty resource file object
  ResFile := TPJResourceFile.Create;
  try
    // Loop thru all source files
    for FileIdx := 0 to Pred(Files.Count) do
    begin
      // Record source file name
      SrcFileName := Files[FileIdx];
      // Get resource name from source name
      ResName := ExtractFileName(SrcFileName);
      // Ensure res name is not a duplicate
      while ResFile.EntryExists(RT_HTML, PChar(ResName), $0809) do
        ResName := '_' + ResName;
      // Create new resource
      Entry := ResFile.AddEntry(RT_HTML, PChar(ResName), $0809);
      // Copy source file into resource data
      SrcStm := TFileStream.Create(SrcFileName, fmOpenRead);
      try
        Entry.Data.CopyFrom(SrcStm, 0);
        Entry.Data.Position := 0;
      finally
        SrcStm.Free;
      end;
    end;
    // Save resource file
    ResFile.SaveToFile(ResFileName);
  finally
    // Free resource file object
    ResFile.Free;
  end;
end;
```

This routine is passed a list of files that are to be included in the resource file (as a string list). The name of the output file is also provided. We first create a new resource file object to store the HTML resources. We then loop through all the files in the list and add a new resource for each file. The resource is named with the name of the file (file name only, no path). To ensure the resource names are not duplicated we repeatedly append underscore characters to duplicate names until they are unique. Having got a unique resource name we create a new entry with the required name and then copy the file's contents into the resource data. Finally we save the resource file and free the resource file object.

Appendix: Resource File Structure Notes

Overview

A 32 bit resource file is comprised as follows:

File Header
Resource 1
Padding
Resource 2
Padding

- - -

Resource N
Padding

File Header

The File header is a "pseudo-resource" that identifies the file as a 32 bit resource file (rather than 16 bit). This is a 32 byte structure, the first 8 bytes of which are $00, $00, $00, $00, $20, $00, $00, $00.

Resource Header

Each resource is made up of a variable length header record followed the resource data. The variable length header is made up of the following fields:

```
DataSize: DWORD;          // size of resource data (excl end padding)
HeaderSize: DWORD;        // size of resource data header
Type: Unicode or Ordinal; // type of resource
Name: Unicode or Ordinal; // name of resource
[Padding: Word];          // optional padding to DWORD boundary
DataVersion: DWORD;       // version of the data resource
MemoryFlags: Word;        // describes the state of the resource
LanguageId: Word;         // language for the resource
Version: DWORD;           // user defined resource version
Characteristics: DWORD;   // user defined info about resource
```

Here is a description of the resource header fields.

Field	Description
DataSize	The size of the resource data that follows the header in bytes, excluding any padding that follows that data.
HeaderSize	The size of the resource header record, including *DataSize* and *HeaderSize* fields.
Type	Variable length field that specifies the resource type either as an ordinal value or as a Unicode string. If the first `Word` of Type is $FFFF the second `Word` is the ordinal value. Otherwise the value is a zero terminated Unicode

	string. Ordinals less than 255 are reserved by Windows. See below for a list of predefined resource types.
Name	Variable length field that specifies the resource name either as an ordinal value or as a Unicode string. If the first `Word` of Name is $FFFF the second `Word` is the ordinal value. Otherwise the value is a zero terminated Unicode string. Note that some resources may not have string names - for example string tables must have ordinal resource ids, all version information resource I have seen have a resource id of 1 and the resource id of an XP manifest is significant to the system.
[Padding]	Optional padding to ensure the following fields begin on a `DWORD` boundary.
DataVersion	Determines the format of the information within the resource header that follows. Often zero. Reserved for use by the system.
MemoryFlags	A bitmask of flags that describe the state of the resource. See the `RES_MF_XXX` flags described in the Constants section for details.
LanguageID	Specifies the language used for any strings in the resource or 0 if language neutral. Bits 0-9 of this `Word` contain the primary language ID while bits 11-15 contain the sub language ID (dialect or variation).
Version	Stores custom version information - sometimes used by the resource compiler. Ignored by the system and stripped out on linking into the application.
Characteristics	Stores custom information about the resource - sometimes used by the resource compiler. Ignored by the system and stripped out on linking into the application.

Predefined Resource Types

The following table lists the predefined resource types known at the time of writing. `RT_HTML` and `RT_MANIFEST` are not defined in Delphi's Windows unit.

Resource Name	Ordinal value	Description
`RT_ACCELERATOR`	9	Accelerator table
`RT_ANICURSOR`	21	Animated cursor
`RT_ANIICON`	22	Animated icon
`RT_BITMAP`	2	Bitmap resource
`RT_CURSOR`	1	Hardware dependent cursor resource
`RT_DIALOG`	5	Dialog box

`RT_DLGINCLUDE`	17	Allows a resource editing tool to associate a string with an .rc file. Typically, the string is the name of the header file that provides symbolic names. The resource compiler parses the string but otherwise ignores the value. For example:
		`/* file foo.dlg */`
		`1 DLGINCLUDE "foo.h"`

`RT_FONT`	8	Font resource
`RT_FONTDIR`	7	Font directory resource
`RT_GROUP_CURSOR`	12	Hardware independent cursor resource - refers to `RT_CURSOR`
`RT_GROUP_ICON`	14	Hardware independent icon resource - refers to `RT_ICON`
`RT_HTML`	23	HTML "Documents": .html, .gif, .css code etc.
`RT_ICON`	3	Hardware dependant icon resource
`RT_MANIFEST`	24	Side-by-side assembly XML manifest (Windows XP)
`RT_MENU`	4	Menu resource
`RT_MESSAGETABLE`	11	Message table entry
`RT_PLUGPLAY`	19	Plug and play resource
`RT_RCDATA`	10	Application or user defined resource - raw data
`RT_STRING`	6	String table entry
`RT_VERSION`	16	Version information
`RT_VXD`	20	VXD

Note that the `RT_XXX` constants are not assigned the ordinal values directly but are set using *MakeIntResource*. So, for example, `RT_VERSION = MakeIntResource(16)`.

Resource Data

Each new resource starts on a `DWORD` boundary, so there may be padding bytes following the end of each resource if the resource data is not a multiple of 4 bytes in length.

The format of the actual resource data depends on the resource type. We do not go into that detail here since this code only provides low-level access to resources in a file and deals only with raw data.

Credits / Bibliography

Various source documents were used in creating this appendix. Key documents were:

- “Win32 Binary Resource Formats” by Floyd Rogers - text file edited and released by Microsoft Developer Support as `ResFmt.txt`.
- “Windows Resource (`.res`) Files” by Ray Lischner, Tempest Software - from a web page that is no longer available.
- “Win32 Resource File Format” by Marco Cocco, - from a web page that is no longer available.